DocsChisel: Adaptive Tool Documentation Optimization Framework for LLM Agents

You Lu*, Kun Zhang*, Bihuan Chen*, Xin Peng*
*College of Computer Science and Artificial Intelligence, Fudan University, China

arXiv:2608.10037v1 [cs.LG] 10 Aug 2026

***Abstract*—Large language models (LLMs) increasingly rely on external tools to accomplish complex real-world tasks, making tool documentation a critical grounding resource for LLM agents. Existing studies mainly focus on improving the tool-use capabilities of LLM agents, while largely treating tool documentation as a fixed input. Although several recent works attempt to optimize tool documentation through rewriting or compression, little is known about how the information contained in tool documentation affects agent performance across different settings.**

To bridge this gap, we conduct a large-scale empirical study on tool documentation for LLM agents. Our study reveals substantial heterogeneity in the information fields provided by existing tool documentation. Moreover, the effectiveness of different information fields is highly dependent on the task domain, LLM backbone, and agent paradigm, indicating that no fixed tool documentation can consistently generalize across diverse agent settings.

Motivated by these findings, we propose DocsChisel, an adaptive tool documentation optimization framework for LLM agents. DocsChisel analyzes failed execution traces of a target LLM agent to identify documentation-related issues, and iteratively optimizes tool documentation by adding, removing, and refining information fields for each tool. We evaluate DocsChisel against two state-of-the-art baselines, *i.e.,* EasyTool and DRAFT. Experimental results show that DocsChisel improves the task success rate of LLM agents by 95.89% over the original tool documentation and by 75.15%, on average, over existing baselines, while incurring limited optimization time and token overhead.

I. Introduction

Large language models (LLMs) [12, 34] have demonstrated strong capabilities across diverse tasks [11, 18, 20], such as code generation, question answering, and data analysis. To execute real-world tasks, LLMs increasingly rely on external tools, *e.g.,* web browsers, code interpreters, and file systems, leading to the emergence of tool-use LLM agents (referred to as LLM agents hereafter) [13, 38, 40, 44, 54, 56]. Emerging LLM agent ecosystems, including agent development platforms [4, 7], tool hosting platforms [37, 43], and agent evaluation suites [27, 53], provide large collections of reusable tools and standardized tool interfaces. These tools are typically accompanied by tool documentation that includes several information fields about their functionalities, invocation constraints, and other details, helping LLM agents understand and invoke tools correctly.

Traditional API documentation has long been studied in software engineering, primarily to improve its readability, completeness, and usability for developers [5, 6, 24, 30, 32, 33, 39, 46, 49]. However, tool documentation for LLM agents differs in both audience and content. The documentation consumers are LLM agents that must infer tool functionality and generate executable tool invocations directly from in-context documentation. Meanwhile, the documentation emphasis shifts from general API understanding to task-oriented tool use, requiring information such as usage conditions, parameter semantics, invocation constraints, and output interpretation. As such documentation guides runtime tool selection and invocation, ambiguity or missing information can lead to incorrect tool invocations and cascading task failures. Thus, tool documentation should be viewed not only as a human-readable reference, but also as a critical grounding resource whose quality heavily affects the task success rate of LLM agents [19, 40, 58].

Literature. Despite the importance of tool documentation, existing studies [8, 22, 25, 27, 41, 45, 47, 55, 60] mainly focus on adding or improving tool-use capabilities of LLM agents from the perspectives of tool construction, tool retrieval, tool invocation, and evaluation. While these studies have significantly advanced the development of LLM agents, they usually treat tool documentation as a fixed given input, without systematically examining how documentation itself affects LLM agent behavior and task success rate. Only a few recent studies [9, 10, 14, 29, 42, 58] explicitly optimize tool documentation for LLM agents. For instance, EasyTool [58] standardizes diverse tool documentation into concise tool instructions with a unified template, thereby reducing the context overhead caused by documentation, whereas DRAFT [42] iteratively refines tool documentation based on outputs of external tools. However, these approaches mostly optimize tool documentation within existing information fields through rewriting, correction, standardization, or compression. They pay less attention to whether the documentation fields themselves are sufficient, redundant, or suitable for different LLM agents.

In practice, tool documentation across LLM agent ecosystems may differ substantially in the information fields it provides. However, it remains unclear whether the utility of these information fields varies across agent settings. For example, do agents powered by less capable LLM backbones benefit more from explicit usage examples? Do multi-agent paradigms place greater demands on usage guidance and invocation constraints? These observations raise a fundamental question, i.e., *what information fields are provided by existing tool documentation, and how do these information fields affect the task success rate of LLM agents under different agent settings?*

Empirical Study. To understand tool documentation for LLM agents and its impact on task success rate, we conduct a large-scale empirical study across 14 tool-use datasets collected from LLM agent ecosystems. Specifically, we investigate their information field composition, and evaluate the task success

TABLE I: Overview of the Collected Tool-Use Datasets

Dataset	#Tools	Description
WorkBench [48]	26	A tool-use LLM agent benchmark with sandboxes, realistic office tasks, databases, and executable tools.
API-Bank [27]	2,211	A comprehensive evaluation benchmark with diverse APIs and multi-turn dialogues for LLM agents.
ToolLLM [40]	16,464	A large-scale suite of real-world APIs for evaluating multi-step tool invocations by LLM agents.
APIBench [38]	1,645	A massive API corpus for studying LLM connectivity to thousands of remote APIs.
ToolAlpaca [50]	426	A simulated dataset with diverse tool APIs and structured tool documentation for training LLMs.
AnyToolBench [8]	13	A benchmark for hierarchical, large-scale tool calling with self-reflective LLM agents.
ToolBench [53]	232	A benchmark for evaluating open-source LLMs on tool manipulation and invocation tasks.
ShortcutsBench [45]	1,414	A large-scale benchmark based on Apple Shortcuts, containing real APIs, user queries, and action sequences.
WildToolBench [57]	1,600	A tool-use benchmark grounded in real-world user behavior patterns and realistic multi-step scenarios.
SWE-bench [23, 54]	15	A software engineering benchmark requiring LLM agents to resolve real GitHub issues via repository tools.
CRMArena [21]	27	A domain-specific benchmark for professional CRM tasks in realistic enterprise environments.
τ-bench [55]	28	A benchmark for interactions among tools, agents and users with domain-specific tools and dialogues.
TooLe [22]	390	A meta-tool benchmark for deciding whether to invoke tools and selecting appropriate tools.
ToolLens [41]	464	A tool-retrieval dataset designed for multi-tool scenarios, emphasizing complete and diverse tool selection.

rate of two different LLM agent paradigms (*i.e.,* ReAct [56] and Multi-Agent [51]) on WorkBench [48], using three different LLM backbones (*i.e.,* GPT-4o [35], GLM-5 [59] and Claude Haiku 4.5 [3]). We design the following research questions.

- **RQ1 Composition Analysis.** What information fields are commonly contained in different tool documentation?
- **RQ2 Impact Analysis.** How do different information fields affect the task success rate of LLM agents across different task domains, LLM backbones, and agent paradigms?

Our results reveal that existing tool documentation in agent ecosystems exhibits substantial heterogeneity in field composition, while different information fields have different impacts on task success rate of LLM agents across task domains, LLM backbones, and agent paradigms. Specifically, we identify 17 information fields, among which only *tool name* and *functionality description* are provided by all 14 datasets, while *usage guidance* and *invocation constraint* appear in only two datasets. Moreover, the same information field may have different or even opposite effects under different agent settings. On average, adding or removing a single information field changes the task success rate of LLM agents by 6.34 percentage points in absolute terms. These findings demonstrate that fixed tool documentation cannot generalize well across different LLM agent settings and should be optimized adaptively.

Our Approach. Motivated by these insights, we propose DocsChisel, an adaptive tool documentation optimization framework for LLM agents. Specifically, given a target LLM agent, DocsChisel first executes representative tool-use queries with the original tool documentation, and collects failed agent execution traces. Then, DocsChisel analyzes these failed traces to identify documentation-related issues, and uses LLMs to iteratively revise tool documentation. Instead of simply rewriting tool documentation into a shorter or more general form, DocsChisel optimizes both the content and structure of tool documentation by adding, removing, or refining information fields for each tool, thereby generating adaptive tool documentation for different LLM agent settings to improve both tool invocation correctness and task success rate.

Evaluation. We implement a prototype of DocsChisel, and conduct experiments on 74 tools across 9 task domains selected from WorkBench [48] and API-Bank [27] to demonstrate its effectiveness and efficiency, compared with two state-of-the-art baselines, *i.e.,* EasyTool [58] and DRAFT [42]. Experimental results demonstrate that DocsChisel consistently improves task success rate under diverse agent settings by 95.89% compared with using the original tool documentation, and achieves an average of 75.15% improvement compared with baselines. Besides, DocsChisel maintains practical optimization efficiency, requiring 12.65 minutes per tool, on average, to optimize the tool documentation for LLM agents.

Contribution. This work makes the following contributions.

- We conduct a systematic empirical study on tool documentation for LLM agents, revealing the varying effectiveness of different information fields across different agent settings.
- We design and implement an adaptive tool documentation optimization framework, DocsChisel, for LLM agents, thereby improving the tool invocation correctness and task success rate of LLM agents.
- We conduct extensive experiments to demonstrate the effectiveness and efficiency of DocsChisel.

II. Empirical Study

To systematically investigate the two research questions introduced in Sec. I, we present a large-scale empirical study on the information field composition of different tool documentation, and evaluate the task success rate of different LLM agent settings with different tool documentation.

A. Study Design

Documentation Collection. We collect tool documentation from 14 representative datasets shown in Table I, covering diverse tool-use scenarios, *e.g.,* workflow execution, mobile automation, and software engineering tasks. We select these datasets based on the following criteria: *(1) providing rich tool collections and corresponding tool documentation, (2) covering different task domains and execution environments, (3) having strong community influence and being widely used in recent studies on LLM agents.* In total, we collect 24,955 tools along with their corresponding tool documentation.

LLM Selection and Agent Paradigm Setting. To evaluate the effectiveness of tool documentation under different LLM agent settings, we select three representative LLMs as agent backbones, *i.e.,* GPT-4o [35], GLM-5 [59], and Claude Haiku 4.5 [3], covering diverse model families widely used in recent LLM agents. We further instantiate two agent paradigms,

i.e., a ReAct agent implemented with LangChain [25], and a Multi-Agent system implemented with AutoGen [51].

Evaluation Metric. We use *task success rate* (TS) to measure whether an LLM agent successfully completes user queries. Given a test suite containing tool-use queries $\mathcal{Q}$, for each query $q_i \in \mathcal{Q}$, we denote its execution outcome as $s(q_i)$. $s(q_i) = 1$ indicates that the task is correctly completed, and $s(q_i) = 0$ otherwise. We compute the task success rate as $\text{TS} = \frac{1}{|\mathcal{Q}|}\sum_{i=1}^{|\mathcal{Q}|}\mathbb{I}(s(q_i) = 1)$, where $\mathbb{I}(\cdot)$ is the indicator function.

RQ Setup. For **RQ1**, we manually inspect the tool documentation from the 14 datasets we collected in Table I. Because these datasets use different formats and inconsistent information field names, three authors independently extracted candidate information fields, and then aligned semantically equivalent information through discussion using unified names following the most commonly used field names in the collected datasets. Disagreements were resolved by consensus to reduce individual bias. Based on the final aligned taxonomy, we report the coverage ratio of each type of information field across datasets.

For **RQ2**, we conduct the empirical evaluation using the tool documentation provided in WorkBench [48], which contains 26 tools from five task domains, *i.e.,* data analysis, email management, calendar management, project management, and customer relationship management (CRM). We modify the tool documentation by removing or adding one information field at a time while keeping all remaining fields unchanged according to the taxonomy derived from **RQ1**. Specifically, if the target information field already exists in the tool documentation, we remove it to measure the impact of missing information. Otherwise, we manually supplement the information field based on the original documentation and dataset-provided metadata, *e.g.,* tool code and correct execution outcome, to measure the impact of additional information. Based on these documentation variants, we conduct controlled experiments to investigate whether the effectiveness of information fields remains consistent across different task domains, LLM backbones, and agent paradigms.

First, to study whether the impact varies across task domains, we use GPT-4o with the ReAct agent, and evaluate documentation variants on the five task domains in WorkBench. Second, to study whether the impact varies across LLM backbones, we focus on the data analysis domain and the ReAct agent, and repeat the experiments with three different LLM backbones, *i.e.,* GPT-4o, GLM-5, and Claude Haiku 4.5. Third, to study whether the impact varies across agent paradigms, we fix the data analysis domain and GPT-4o, and compare the results under the ReAct and Multi-Agent paradigms. In all experiments, we keep the query set, prompt template, and execution environment unchanged within each comparison.

B. Composition Analysis (RQ1)

After manually analyzing the collected tool documentation, we observe that tools within the same dataset usually follow a consistent documentation convention, *i.e.,* tools in the same dataset tend to contain the same set of information fields. However, such conventions vary across datasets, leading to different field compositions in existing tool documentation.

TABLE II: Prevalence of Information Fields across Datasets

Information Field	Brief Description	Prevalence
Tool Name (TN)	Invocation identifier of target tool	14/14
Functionality Description (FD)	Tool purpose and capability	14/14
Task Domain (TD)	Application scenario or category	6/14
Input Parameter Name (IPN)	Input argument identifier	13/14
Input Parameter Description (IPD)	Input argument semantics	12/14
Input Parameter Type (IPT)	Input data types or formats	13/14
Default Parameter Value (DPV)	Default input parameter values	3/14
Parameter Required Flag (PRF)	Required argument indicator	10/14
Parameter Optional Flag (POF)	Optional argument indicator	7/14
Return Parameter Name (RPN)	Output argument identifier	4/14
Return Parameter Description (RPD)	Output argument semantics	5/14
Return Parameter Type (RPT)	Output data types or formats	5/14
Code Implementation (CI)	Tool implementation snippets	5/14
Usage Example (UE)	Concrete invocation examples	4/14
Response Template (RT)	Expected tool response format	4/14
Usage Guidance (UG)	Tool usage recommendations	2/14
Invocation Constraint (IC)	Invocation conditions or restrictions	2/14

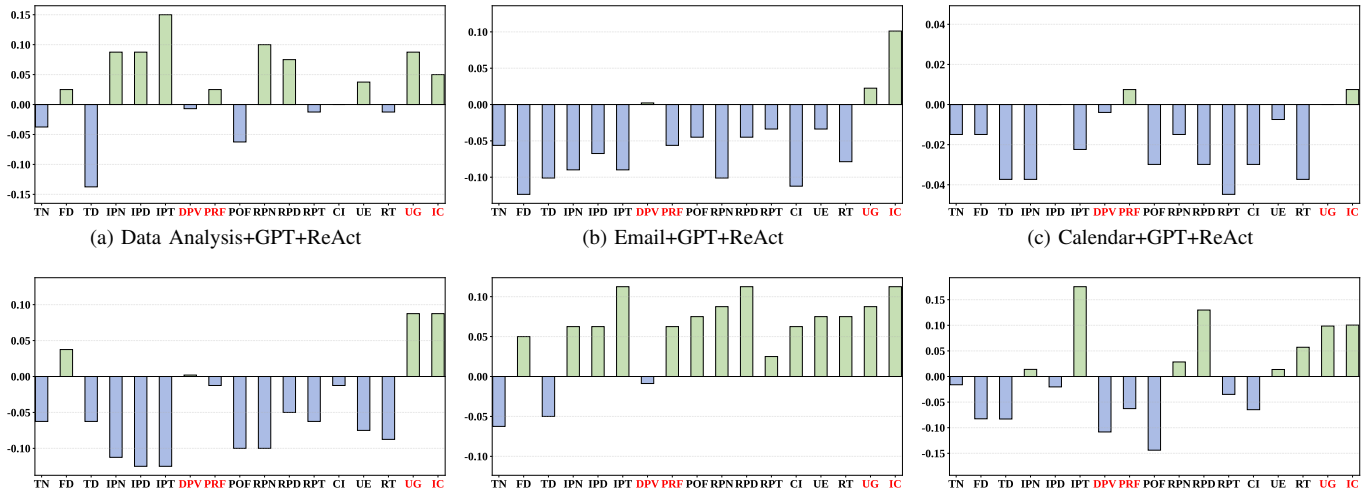
We identify 17 common information fields, and report their prevalence across the 14 datasets. As shown in Table II, fields describing tool identity and input schemas are widely adopted. *Tool name* and *functionality description* appear in all datasets, while *input parameter name*, *input parameter description*, and *input parameter type* appear in most datasets. In contrast, fields supporting task-oriented tool use and result interpretation, such as *default parameter value*, *return parameter name*, *return parameter type*, *usage example* and *response template*, are provided much less consistently. *Usage guidance* and *invocation constraint* appear in only two datasets.

Findings. Existing tool documentation shows cross-dataset heterogeneity in information field composition. Although tools within the same dataset follow a consistent convention, different datasets provide different sets of information fields.

C. Impact Analysis (RQ2)

Fig. 1 reports the change in TS after adding or removing one information field from the original tool documentation. Black x-axis labels denote removed fields, while red labels denote added fields. As shown in Fig. 1a-1c, under GPT-4o with ReAct, the evaluated fields exhibit both consistent and domain-dependent effects. Among the 17 fields, only 6 fields show the same effect direction across all three domains. Removing *tool name* (TN), *task domain* (TD), *return parameter type* (RPT), *response template* (RT), or *parameter optional flag* (POF) consistently decreases TS, whereas adding *invocation constraint* (IC) consistently improves it. For example, removing TD decreases TS by 3.75 to 13.75 percentage points, while adding IC improves it by 0.75 to 11.25 percentage points. In contrast, the remaining fields show domain-dependent effects. Adding *parameter required flag* (PRF), for instance, changes TS by +2.50, -5.61, and +0.75 percentage points in data analysis, email management, and calendar management, respectively. Due to space limitations, results for project management and customer relationship management are provided on our website [1].

The effects also vary across LLM backbones. As shown in Fig. 1a, Fig. 1d, and Fig. 1e, when fixing the domain and agent paradigm, 12 of the 17 fields exhibit different effect directions across GPT-4o, GLM-5, and Claude Haiku 4.5. For example, removing POF decreases TS for GPT-4o and GLM-5, but improves it by 7.5 percentage points for Claude Haiku 4.5. This indicates that different LLM backbones rely on different



(a) Data Analysis+GPT+ReAct (b) Email+GPT+ReAct (c) Calendar+GPT+ReAct

(d) Data Analysis+GLM+ReAct (e) Data Analysis+Claude+ReAct (f) Data Analysis+GPT+Multi-Agent

Fig. 1: Impact of Information Fields on Task Success Rate across Task Domains, LLM Backbones, and Agent Paradigms

documentation information. A similar variation is observed across agent paradigms. As shown in Fig. 1a and Fig. 1f, replacing ReAct with Multi-Agent reverses the effects of several fields, including *functionality description* (FD), *input parameter description* (IPD), *parameter required flag* (PRF), and RT. Therefore, although some fields provide stable benefits or harms, the effectiveness of fields depends on the task domain, LLM backbone, and agent paradigm.

Findings. Different information fields in tool documentation have different impacts on task success rate of LLM agents across task domains, LLM backbones, and agent paradigms.

D. Empirical Insights

The empirical study provides two insights for tool documentation optimization. First, no fixed documentation convention can consistently generalize across different LLM agent settings, since the effectiveness of information fields may vary across task domains, LLM backbones, and agent paradigms. Second, an effective tool documentation optimizer should adaptively adjust the information field composition based on the LLM agent setting according to different execution feedback.

III. Methodology

Motivated by the empirical insights in Sec. II, we propose DocsChisel, an adaptive framework for optimizing tool documentation for LLM agents. Given a target LLM agent, a set of tools with their original documentation, and a set of queries with execution ground truth, DocsChisel generates optimized tool documentation for each tool under the target agent setting. The key idea is to perform tool documentation optimization at the information-field level for each tool under the guidance of failed execution traces of the target agent.

A. Approach Overview

Fig. 2 shows the approach overview. First, DocsChisel performs domain-aware execution profiling with the original tool documentation (Sec. III-B). In this stage, when the target

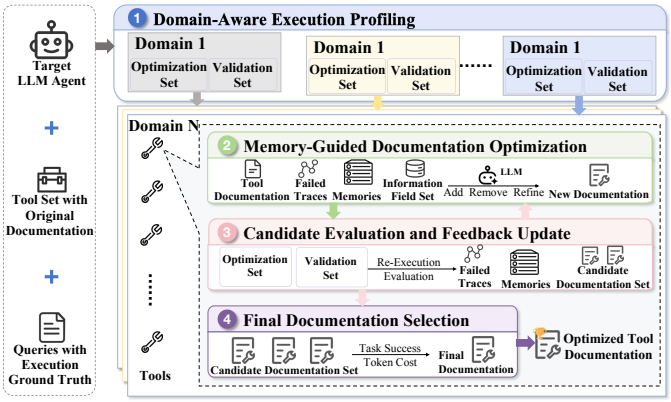


Fig. 2: Approach Overview of DocsChisel

agent is equipped with tools from multiple task domains, DocsChisel organizes the tools and queries by task domain. Within each domain, it associates each tool with the queries whose execution requires that tool, and partitions these tool-associated queries and their execution ground truth into optimization and validation sets. It then executes the target agent to collect traces on each tool's optimization set and obtain its initial task success rate on the corresponding validation set.

Then, DocsChisel enters an iterative optimization loop for each tool in the current task domain. At each iteration, DocsChisel performs memory-guided documentation optimization for the target tool using the queries associated with that tool in the optimization set (Sec. III-C). Given the original tool documentation, the latest failed execution traces collected in the optimization set, the field optimization memories of the current domain, and the information field set, DocsChisel uses an LLM-based exploration process to generate new tool documentation, including adding missing information fields, removing distracting or redundant fields, and refining ambiguous or incomplete fields. Each generated tool documentation is then evaluated through agent re-execution to update the feedback (Sec. III-D). DocsChisel first executes

the target agent with the generated tool documentation on the optimization set to obtain new failed traces, and updates the field optimization memories for the next round of tool documentation generation. Then, it evaluates the generated tool documentation on its validation set to update the candidate documentation set. Once the maximum number of optimization iterations for the target tool is reached, DocsChisel selects the best documentation from its candidate set (Sec. III-E). It then repeats the optimization process for the next tool until all tools in the current domain have been processed.

B. Domain-Aware Execution Profiling

Our empirical study shows that different task domains rely on different information fields in tool documentation. Meanwhile, a user query may require the agent to invoke multiple tools within the same domain to complete the task, making the effectiveness of one tool's documentation related to other semantically relevant tools. Therefore, we first organize tools and queries by task domain, so that subsequent documentation optimization can be performed within each domain and exploit the domain-specific information field optimization experience.

For task domain c, let $\mathcal{T}_c$ denote the tools in this domain and let $\mathcal{D}_c^0 = \{d_t^0 \mid t \in \mathcal{T}_c\}$ denote the tool documentation set composed of original documentation d_t^0 for each tool t. Let $\mathcal{Q}_{c,t}$ denote the domain-associated queries whose ground-truth executions involve tool t. A query involving multiple tools belongs to multiple tool-specific query sets. We partition each $\mathcal{Q}_{c,t}$ into an optimization set $\mathcal{Q}_{c,t}^{opt}$ and a validation set $\mathcal{Q}_{c,t}^{val}$. We further denote the collections of tool-specific optimization and validation query sets in domain c as $\mathcal{Q}_c^{opt} = \{\mathcal{Q}_{c,t}^{opt} \mid t \in \mathcal{T}_c\}$ and $\mathcal{Q}_c^{val} = \{\mathcal{Q}_{c,t}^{val} \mid t \in \mathcal{T}_c\}$, respectively.

Then, we execute the target LLM agent on both sets using the original tool documentation $\mathcal{D}_c^0$. For queries in $\mathcal{Q}_{c,t}^{opt}$, we collect execution traces, which include the user query, the reasoning steps, invoked tools, tool inputs, tool outputs, and available error messages. Let $\mathcal{F}_c^0 = \{\mathcal{F}_{c,t}^0 \mid t \in \mathcal{T}_c\}$ and $\mathcal{S}_c^0 = \{\mathcal{S}_{c,t}^0 \mid t \in \mathcal{T}_c\}$ denote the failed and successful execution trace collections obtained using the original tool documentation $\mathcal{D}_c^0$, where $\mathcal{F}_{c,t}^0$ and $\mathcal{S}_{c,t}^0$ represent the failed and successful traces involving tool t, respectively. The failed traces are used as evidence for optimizing tool documentation, while the successful traces are retained for regression checking. For queries in validation sets, we record the initial task success rate $\mathrm{TS}_c^0 = \{\mathrm{TS}_{c,t}^0 \mid t \in \mathcal{T}_c\}$ of the target LLM agent as a reference before documentation optimization. Finally, the output of this stage is a profiling result $\langle \mathcal{T}_c, \mathcal{D}_c^0, \mathcal{Q}_c^{opt}, \mathcal{Q}_c^{val}, \mathcal{F}_c^0, \mathcal{S}_c^0, \mathrm{TS}_c^0 \rangle$ for each task domain.

C. Memory-Guided Documentation Optimization

Within each task domain c, DocsChisel optimizes the original tool documentation d_t^0 of each tool $t \in \mathcal{T}_c$ iteratively. At iteration n, instead of directly asking an LLM to rewrite the current documentation [9, 58], DocsChisel decomposes the optimization process into three LLM-assisted steps, i.e., *failed trace diagnosis*, *field operation planning*, and *optimized documentation generation*, as illustrated in Fig. 3. The diagnosis step identifies why the LLM agent fails under the current

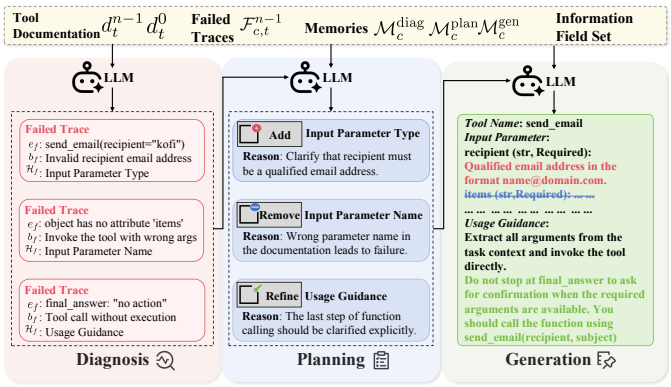


Fig. 3: Memory-Guided Documentation Optimization

Failed Trace Diagnosis

#Task Prompt:

Your task is to diagnose failed execution traces for tool documentation optimization. Given the current tool documentation: {Current Documentation}, the failed trace: {Failed Trace}, identify why the agent fails and attribute it to possible information fields: {Information Field List}.
Return a diagnosis record in the form:
{"Failure Evidence": " ", "Behavior Mismatch": " ", "Related Fields": []}

#Memory Mechanisms:

Below are recurring failure patterns in this domain: {Diagnosis Memory}
Use them to distinguish isolated failures from documentation-related issues.

Fig. 4: Prompt Template for Failed Trace Diagnosis

documentation. The planning step maps the diagnosed issues to possible field-level operations. The generation step applies the planned operations to produce new tool documentation d_t^n.

Failed Trace Diagnosis. At iteration n, DocsChisel first diagnoses the failed execution traces $\mathcal{F}_{c,t}^{n-1}$ collected from optimization set $\mathcal{Q}_{c,t}^{opt}$ under the latest documentation d_t^{n-1}. Since execution failures may be caused by factors beyond tool documentation, we prompt an LLM to inspect each failed trace together with the current documentation d_t^{n-1}, the ground truth, and the diagnosis memory $\mathcal{M}_c^{\text{diag}}$ of the current task domain.

The diagnosis memory stores domain-specific failure patterns observed in previous iterations, including frequently occurring agent errors and their potentially related information fields. By referring to such memory, the LLM can reuse diagnosis experience accumulated from previous iterations, rather than analyzing each failed trace in isolation. Fig. 4 shows the template of the diagnosis prompt for the LLM. Specifically, the LLM is required to summarize the failure evidence, identify the mismatch between agent behaviors and expected tool invocations, and infer the potentially responsible information fields from those summarized in Table II for failure cases.

Formally, for each failed trace $f \in \mathcal{F}_{c,t}^{n-1}$, we obtain a diagnosis record $r_f = \langle e_f, b_f, \mathcal{H}_f \rangle$, where e_f denotes the summarized failure evidence, b_f denotes the agent behavior mismatch with respect to the ground truth, and $\mathcal{H}_f$ denotes the set of potentially related information fields. The diagnosis records of all failed traces form the set $\mathcal{R}_t^n = \{r_f \mid f \in \mathcal{F}_{c,t}^{n-1}\}$, which is then passed to the field operation planning step.

Field Operation Planning. After obtaining the diagnosis set $\mathcal{R}_t^n$, DocsChisel further plans how to revise the tool documentation at the information-field level. Given $\mathcal{R}_t^n$, the

Field Operation Planning

#Task Prompt:
Your task is to plan operations for revising the tool documentation.
Given the diagnosis records: {Diagnosis Records}, the original documentation: {Original Documentation}, decide which fields should be added, removed, or refined.
Return an operation plan in the form:
{"Operation": "Add/Remove/Refine", "Target Field": " ", "Reason": " "}

#Memory Mechanisms:
Below are historical operation experiences in this domain: {Planning Memory}
Use them to merge duplicated operations and avoid conflicting revisions.

Fig. 5: Prompt Template for Field Operation Planning

original documentation d_t^0, and the planning memory $\mathcal{M}_c^{\text{plan}}$ of the current task domain, we prompt an LLM to map the diagnosed issues to concrete information field operations. Here, d_t^0 serves as the semantic anchor for planning operations, preventing the optimization process from drifting away from the original tool documentation semantics.

The planning memory stores domain-specific operation experience observed in previous iterations, including which field operations are useful for addressing certain failure patterns. For example, missing invocation conditions can be mapped to adding invocation constraints, distracting usage examples or implementation snippets can be mapped to removing redundant fields, and confusion between similar tools can be mapped to refining functionality descriptions or usage guidance. By referring to such memory, DocsChisel avoids planning field operations from each failed trace in isolation, and instead exploits historical optimization experience from related tools in the same domain. Fig. 5 shows the template of the planning prompt for the LLM. Specifically, the LLM is required to examine each diagnosis record in $\mathcal{R}_t^n$, determine whether the diagnosed issue requires adding, removing, or refining an information field, and provide the rationale for the operation.

Formally, for each diagnosis record $r_f \in \mathcal{R}_t^n$, we obtain an information-field operation plan $p_f = \{\langle o_f, h_f, g_f\rangle \mid h_f \in \mathcal{H}_f\}$, where $o_f \in \{\text{ADD}, \text{REMOVE}, \text{REFINE}\}$ denotes the planned operation, $h_f \in \mathcal{H}_f$ denotes the target information field, and g_f denotes the reason for applying this operation. Since different failed traces may lead to duplicated or conflicting operations, we merge trace-level plans into a unified operation plan $\mathcal{P}_t^n$ by removing duplicated operations and resolving conflicts according to their supporting diagnosis records, the planning memory, and the original documentation. The resulting plan $\mathcal{P}_t^n$ specifies which information fields should be added, removed, or refined in the next documentation generation step.

Optimized Documentation Generation. Given the unified operation plan $\mathcal{P}_t^n$, DocsChisel generates new tool documentation for tool t, using the original documentation d_t^0 as the generation anchor to preserve the original tool semantics and reduce the risk of information drift during repeated rewriting.

To guide the generation process, DocsChisel further incorporates the generation memory $\mathcal{M}_c^{\text{gen}}$ of the current task domain. The generation memory stores domain-specific editing experience observed in previous iterations, including field expressions that are effective for mitigating certain failure patterns, generation constraints, and edits that may introduce

Optimized Documentation Generation

#Task Prompt:
Your task is to generate optimized tool documentation.
Given the original documentation: {Original Documentation} and the unified operation plan: {Operation Plan}, apply the planned field-level operations to produce new documentation. Preserve the original documentation as much as possible, and only add, remove, or refine fields specified by the operation plan.
Return the result in the form:
{"Optimized Documentation": " "}

#Memory Mechanisms:
Below are effective editing experiences in this domain: {Generation Memory}
Use them to improve field expressions and avoid regression errors.

Fig. 6: Prompt Template for Documentation Generation

Memory Update

#Task Prompt:
Your task is to update domain-level optimization memories.
Given the new documentation: {New Documentation}, the operation plan: {Operation Plan}, the validation result: {Validation Result}, and the regression traces: {Regression Traces}, summarize reusable experience from this iteration. Identify newly observed failure patterns, effective field operations, and editing rules that may help future tools in the same domain. Return the result in the form:
{"Diagnosis Memory": [], "Planning Memory": [], "Generation Memory": []}

#Memory Mechanisms:
Below are previous memories in this domain: {Previous Memories}
Revise them with the new evidence and avoid preserving suggestions that caused regression errors.

Fig. 7: Prompt Template for Updating the Memory

regression errors. Fig. 6 shows the template of the generation prompt for the LLM. Specifically, the LLM is required to apply the operations in $\mathcal{P}_t^n$ to d_t^0 by adding missing fields, removing distracting or redundant fields, and refining ambiguous or incomplete fields, while preserving unchanged information that is not targeted by the operation plan.

D. Candidate Evaluation and Feedback Update

After generating the candidate documentation d_t^n, we evaluate whether it improves the task success rate of the target LLM agent through agent re-execution, and update the optimization feedback for subsequent iterations. Since the optimized documentation may improve some failed queries while harming previously successful ones, we consider both validation performance and regression risk during the evaluation.

Regression-Aware Evaluation. For the target tool t, we replace its original documentation d_t^0 with the new documentation d_t^n while keeping the documentation of other tools unchanged. Then, we execute the LLM agent on $\mathcal{Q}_{c,t}^{val}$ and compute the task success $\text{TS}_{c,t}^n$. In addition, we check whether d_t^n introduces regression errors on previously successful optimization queries by re-executing the LLM agent on $\mathcal{Q}_{c,t}^{opt}$ to obtain the failed trace set $\mathcal{F}_{c,t}^n$ and the successful trace set $\mathcal{S}_{c,t}^n$. Based on these results, we obtain the regression set $\mathcal{G}_{c,t}^n$, which contains the failed traces of queries that were successfully completed under the original documentation d_t^0 but failed under d_t^n.

Candidate and Memory Update. We compare the newly generated documentation d_t^n with the original documentation d_t^0 using the validation task success rate $\text{TS}_{c,t}^0$ obtained during execution profiling. Only when $\text{TS}_{c,t}^n \geq \text{TS}_{c,t}^0$, we add the new tool documentation d_t^n to the candidate documentation set $\mathcal{D}_c^t$.

Meanwhile, the failed traces $\mathcal{F}_{c,t}^n$ collected on $\mathcal{Q}_{c,t}^{opt}$ are used as the input for failed trace diagnosis in the next iteration.

We further update the domain-level memory based on the evaluation results. Specifically, we prompt an LLM to analyze the new documentation d_t^n, the regression set $\mathcal{G}_{c,t}^n$ and the validation result $\text{TS}_{c,t}^n$. Fig. 7 shows the prompt template for the LLM. Specifically, we require the LLM to summarize the newly observed failure patterns, effective field operations and editing experience during this optimization iteration. Then, we update the $\mathcal{M}_c^{\text{diag}}$, $\mathcal{M}_c^{\text{plan}}$ and $\mathcal{M}_c^{\text{gen}}$, respectively. In this way, the updated traces and memories provide feedback for the next iteration, forming a closed optimization loop.

E. Final Documentation Selection

When optimization for tool t reaches the maximum number of iterations, we select the candidate with the highest validation task success rate from $\mathcal{D}_c^t$, breaking ties by choosing the shortest documentation to reduce execution context cost. We then proceed to the next tool until all tools in $\mathcal{T}_c$ are processed.

IV. Evaluation

We implement a prototype of DocsChisel with 2,687 lines of Python code. To evaluate the effectiveness and efficiency of DocsChisel, we design the following research questions.

- **RQ3 Effectiveness Evaluation.** What is the effectiveness of DocsChisel in improving task success rate of LLM agents?
- **RQ4 Efficiency Evaluation.** What is the efficiency and optimization overhead of DocsChisel?
- **RQ5 Sensitivity Analysis.** How do optimization models and iteration budgets affect the effectiveness of DocsChisel?
- **RQ6 Ablation Study.** What is the contribution of our memory mechanism to the effectiveness of DocsChisel?

A. Evaluation Setup

Dataset. We select two tool-use benchmarks, *i.e.,* WorkBench [48] and API-Bank [27]. In addition to the five task domains from WorkBench used in **RQ2**, API-Bank provides tools from four additional domains, *i.e.,* finance management, health management, information retrieval, and smart home management (SHM). Overall, our evaluation dataset covers 9 task domains, involving 74 tools and 2,072 user queries. We split the user queries into optimization, validation, and test sets with a ratio of 5:1:4. The optimization and validation sets are used during documentation optimization, while the test set is held out throughout the optimization process and used only for the final evaluation of the optimized tool documentation. To ensure a fair and representative evaluation, we perform a stratified split over task domains and tools, so that each subset preserves the overall distribution of tool-use scenarios as much as possible. After splitting, the optimization, validation, and test sets contain 1,036, 207, and 829 user queries, respectively. The detailed statistics of the evaluation dataset across task domains and tools are available at [1] due to space limitations.

Metrics. Beyond the TS metric used in our empirical study, we also use *tool invocation correctness* (TC) to measure whether the LLM agent invokes the correct tools. Specifically, given the test suite containing tool-use queries $\mathcal{Q}$, for each query $q_i \in \mathcal{Q}$, we denote the ground-truth tool set as T_{q_i} and the tool set predicted by the LLM agent as $\hat{T}_{q_i}$. A query is regarded as tool-correct only when the predicted tool set exactly matches the ground-truth set, *i.e.,* $T_{q_i} = \hat{T}_{q_i}$. We compute tool invocation correctness as $\text{TC} = \frac{1}{|\mathcal{Q}|}\sum_{i=1}^{|\mathcal{Q}|} \mathbb{I}(T_{q_i} = \hat{T}_{q_i})$, where $\mathbb{I}(\cdot)$ is the indicator function. Together, TC and TS capture the effectiveness of tool documentation on both tool invocation behavior and task completion performance of LLM agents.

Baseline. We compare DocsChisel with two tool documentation optimization approaches, *i.e.,* EasyTool [58] and DRAFT [42]. EasyTool uses an LLM and a predefined template to convert documentation into concise, structured tool-use instructions. DRAFT uses LLMs to iteratively refine tool documentation based on feedback obtained by generating tool inputs and executing the tools. We use their official artifacts and adapt only the input interfaces to our dataset.

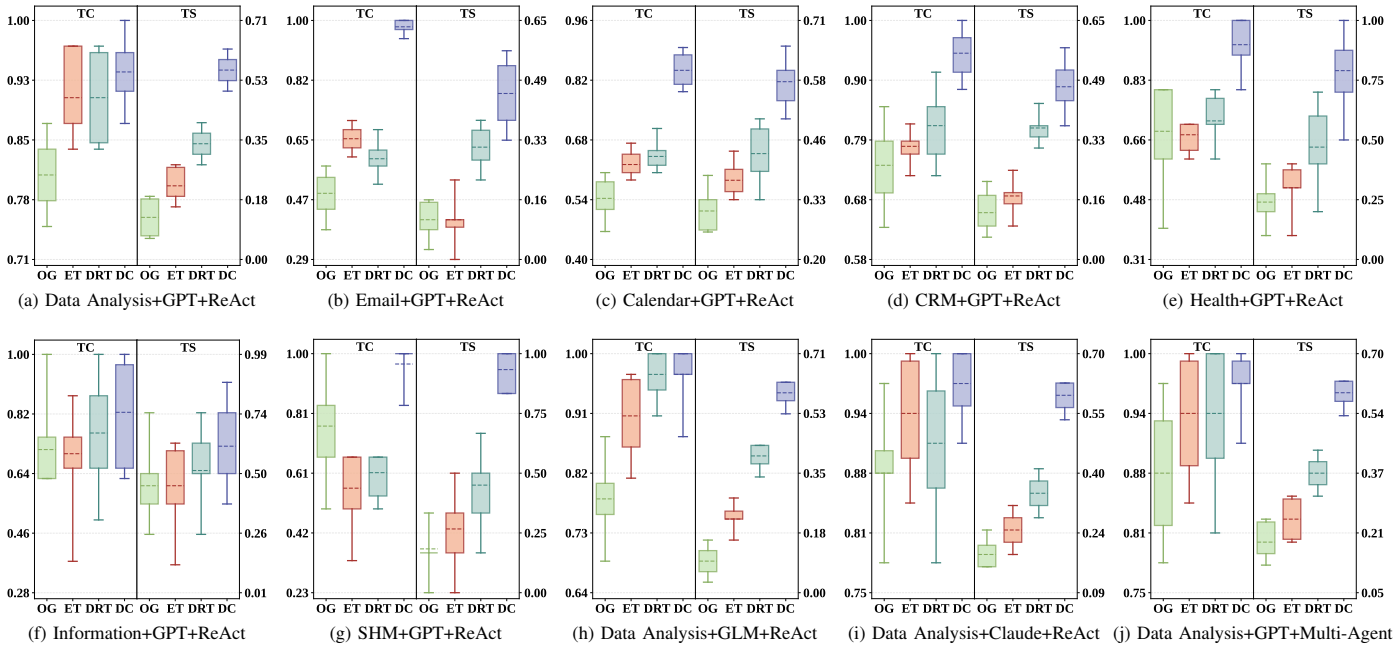
RQ Setup. For **RQ3** and **RQ4**, we apply DocsChisel with Claude Haiku 4.5 as the optimization model to optimize the original tool documentation in our dataset. We set the maximum number of iterations to 5 by default to balance optimization effectiveness and LLM token overhead. For **RQ3**, we evaluate the effectiveness of DocsChisel by comparing it with using the original tool documentation and using the tool documentation optimized by EasyTool and DRAFT, respectively. To understand whether the improvement generalizes across different agent settings, we conduct comparisons from three perspectives introduced in **RQ2**, *i.e.,* task domains, LLM backbones, and agent paradigms, respectively.

For **RQ4**, we report the additional token cost introduced by the optimized documentation during agent execution, compared with the original documentation. We also measure the optimization overhead of DocsChisel, including the time cost and token consumption required to generate optimized tool documentation, and compare them with the baselines.

For **RQ5**, we examine how different optimization models and iteration budgets affect DocsChisel in the data analysis domain. We vary the LLM used for documentation optimization, *i.e.,* GPT-4o and GLM-5, and the maximum number of optimization iterations, while keeping the original tool documentation, query set, and target agent unchanged. The target agent is a GPT-4o-based ReAct agent. We report the average TC and TS on the test set using the optimized documentation generated under each setting.

For **RQ6**, we conduct an ablation study to evaluate the contribution of our memory mechanism. We construct a variant of DocsChisel, denoted as DocsChisel-NoMem, by disabling the memory mechanism while keeping the remaining optimization pipeline unchanged. We then compare the tool documentation in the data analysis domain optimized by DocsChisel and DocsChisel-NoMem, and report the average TC and TS of the GPT-4o-based ReAct agent, analyzing whether the memory mechanism helps DocsChisel produce more effective tool documentation for the target LLM agent.

Environment. We conduct all the experiments on Ubuntu 20.04.4 LTS servers with 4 NVIDIA GeForce RTX 3090 GPUs, Intel(R) Xeon(R) Silver 4310 @ 2.10GHz and 128GB memory.



(a) Data Analysis+GPT+ReAct (b) Email+GPT+ReAct (c) Calendar+GPT+ReAct (d) CRM+GPT+ReAct (e) Health+GPT+ReAct

(f) Information+GPT+ReAct (g) SHM+GPT+ReAct (h) Data Analysis+GLM+ReAct (i) Data Analysis+Claude+ReAct (j) Data Analysis+GPT+Multi-Agent

Fig. 8: Effectiveness Comparison across Task Domains, LLM Backbones, and Agent Paradigms

B. Effectiveness Evaluation (RQ3)

Overall Results. Fig. 8 compares DocsChisel (DC) with the original documentation (OG), EasyTool (ET), and DRAFT (DRT) across task domains, LLM backbones, and agent paradigms. We independently repeat the optimization process three times. For each optimized result, we conduct five independent evaluation runs on the test set to account for execution randomness. Hence, each boxplot summarizes the results from $3 \times 5 = 15$ runs. Overall, DocsChisel achieves the highest average TC and TS in all evaluated settings. Compared with the original documentation, DocsChisel improves TC by 34.69% and TS by 95.89% on average. Compared with the two baseline approaches, it achieves an average TC improvement of 30.83% and TS improvement of 75.15%. Even against the strongest baseline in each setting, DocsChisel improves TC and TS by 26.23% and 46.11% on average, respectively.

With respect to task domains, Fig. 8a-8g show that DocsChisel consistently achieves the highest average TC and TS in the seven domains presented in the paper. The remaining two domains, *i.e.,* project management and finance management, are reported on our website [1] due to space limitations. Across all nine evaluated domains, DocsChisel improves TC and TS by 46.27% and 194.22% over the original documentation, respectively, and by 27.34% and 42.36% over the strongest documentation optimization baseline on average. The gains are particularly evident in domains such as email management, customer relationship management, health management, and smart home management, where the original and baseline documentation often yields low task success rate despite moderate tool invocation correctness.

With respect to LLM backbones, Fig. 8a, Fig. 8h, and Fig. 8i show that DocsChisel remains best-performing for all three backbones. Compared with the strongest baseline, it improves TC by 3.45% and TS by 60.83%, on average, across GPT-4o, GLM-5, and Claude Haiku 4.5.

With respect to agent paradigms, Fig. 8a and Fig. 8j compare ReAct and Multi-Agent under the same task domain and GPT-4o backbone. DocsChisel achieves the highest TC and TS under both paradigms. Compared with the strongest baseline, it improves TC by 3.44% and 3.34%, and TS by 63.61% and 58.35% under ReAct and Multi-Agent, respectively.

The results further show that DocsChisel generally achieves higher lower-end performance and more compact distributions across repeated runs. Compared with EasyTool and DRAFT, DocsChisel reduces the average interquartile range of TC by 41.56% and 52.93%, respectively, and that of TS by 36.71% and 44.07%. These results indicate that DocsChisel introduces stable improvements across different agent settings.

Breakdown Analysis. EasyTool improves some settings by transforming documentation into concise and structured instructions. However, its fixed template applies the same information fields across agent settings. It may remove fields useful to a particular agent or fail to introduce fields absent from the original documentation, resulting in limited improvements.

DRAFT uses feedback obtained by generating tool inputs and executing tools to iteratively refine tool documentation. Although such feedback helps correct tool descriptions, it mainly reflects isolated tool executions rather than failures occurring during end-to-end agent task solving. Moreover, it does not explicitly adapt the composition of information fields. Consequently, documentation-related failures observed in complete agent traces may remain unresolved.

In contrast, DocsChisel diagnoses failed agent execution traces and performs field-level addition, removal, and refinement according to the target agent setting, allowing for

TABLE III: Results of Efficiency Evaluation

Approach	Doc Length	Token Cost	Time Cost (min)
Original	152.58	–	–
EASYTOOL	82.68	237.81	0.23
DRAFT	189.56	4,784.39	2.26
DOCSCHISEL	189.23	4,480.81	12.65

TABLE IV: Results of Model Sensitivity

Metrics	GPT-4o	GLM-5	Claude Haiku 4.5
TC	47.50%	73.75%	93.75%
TS	21.25%	35.63%	56.25%

better alignment of tool documentation with the information requirements of the target agent, explaining its consistent improvements in both TC and TS. Nevertheless, DOCSCHISEL does not achieve perfect results. Some failures originate from factors beyond tool documentation, such as incorrect task decomposition, stochastic LLM reasoning, or incomplete user queries. Therefore, documentation optimization can substantially improve agent effectiveness, but cannot eliminate failures caused by other components of the agent execution process.

Summary. DOCSCHISEL consistently outperforms EASYTOOL and DRAFT across task domains, LLM backbones, and agent paradigms, improving tool invocation correctness by 30.83% and task success rate by 75.15%, on average.

C. Efficiency Evaluation (RQ4)

Table III reports the average documentation length and optimization overhead per tool. EASYTOOL produces the shortest documentation because it primarily compresses the original content. In contrast, DRAFT and DOCSCHISEL generate slightly longer documentation by incorporating additional information. The documentation produced by DOCSCHISEL contains 189.23 tokens on average, which is 24.02% longer than the original documentation but comparable to DRAFT. Thus, DOCSCHISEL introduces only a limited additional context cost.

For optimization overhead, DOCSCHISEL consumes 6.35% fewer tokens than DRAFT for each tool during optimization. However, it requires 12.65 minutes on average, longer than EASYTOOL and DRAFT, because DOCSCHISEL repeatedly diagnoses failed traces, generates candidate documentation, and evaluates candidates through agent re-execution. Overall, DOCSCHISEL trades additional offline optimization time for substantially improved effectiveness, while maintaining comparable documentation length and token consumption to DRAFT.

Summary. DOCSCHISEL incurs a higher optimization-time cost, requiring 12.65 minutes per tool on average, while consuming 6.35% fewer optimization tokens than DRAFT and producing documentation of comparable length.

D. Sensitivity Analysis (RQ5)

Model Sensitivity. Table IV reports the effectiveness of DOCSCHISEL using different optimization models while keeping the target agent and evaluation setting unchanged. Claude Haiku 4.5 achieves the best performance, improving TC and TS over GLM-5 by 27.12% and 57.87%, respectively, and over GPT-4o by 97.37% and 164.71%. GLM-5 also outperforms GPT-4o, with relative gains of 55.26% in TC and 67.67% in TS. These results indicate that the quality of optimized documentation is sensitive to the optimization model. Documentation optimization requires the model to diagnose failed traces, identify field-level deficiencies, and generate appropriate revisions. Models with stronger instruction-following and reasoning capabilities can produce more effective documentation.

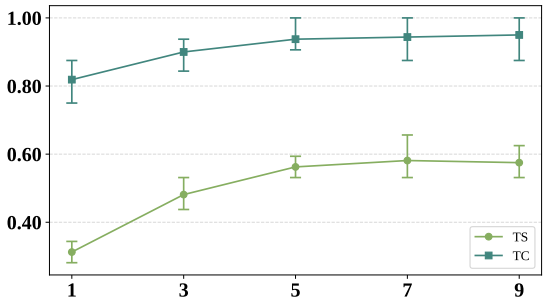


Fig. 9: Results of Iteration Sensitivity

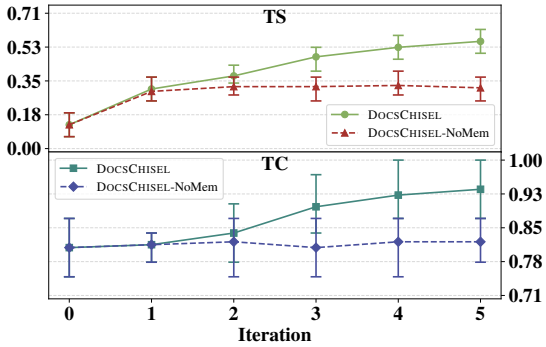


Fig. 10: Results of Ablation Study

Iteration Sensitivity. Fig. 9 shows that increasing the number of optimization iterations improves both TC and TS. The gains are most pronounced in the early rounds. Compared with one iteration, five iterations improve TS by 80.00% and TC by 14.50%. This indicates that iterative execution feedback progressively helps DOCSCHISEL identify and address documentation deficiencies. After five iterations, however, the marginal gains become small, suggesting that the optimization process has largely converged. We therefore set the default iteration budget to five to avoid large optimization overhead.

Summary. Stronger optimization models generally yield better documentation quality, while iterative refinement substantially improves optimization effectiveness. The budget of five iterations achieves a favorable trade-off.

E. Ablation Study (RQ6)

Fig. 10 compares DOCSCHISEL with DOCSCHISEL-NoMem over five optimization iterations. In the first two iterations, the two variants achieve similar performance, indicating that both can improve documentation using the feedback from the current iteration. However, their performance diverges as optimization proceeds. At the fifth iteration, DOCSCHISEL-NoMem reduces TC and TS by 13.64% and 76.47% compared with DOCSCHISEL, respectively. Moreover, the relative TS improvement

between DocsChisel and DocsChisel-NoMem increases from 4.17% at Iteration-1 to 76.47% at Iteration-5, indicating that the benefit of memory becomes more pronounced as optimization experience accumulates. Without memory, both metrics largely plateau after the second iteration. In contrast, DocsChisel continues to improve by reusing previously observed documentation deficiencies, effective field operations, and editing experience. Such experience guides subsequent diagnosis, field planning, and candidate generation, preventing the optimizer from repeatedly exploring ineffective revisions.

Summary. The memory mechanism is critical to iterative documentation optimization, improving the final TC and TS, on average, by 13.64% and 76.47%, respectively.

V. Threats to Validity

First, the selection of agent paradigms, LLM backbones, and datasets may limit the generalizability of our findings. To mitigate this threat, we evaluate DocsChisel using two agent paradigms with three LLM backbones across nine task domains.

Second, our evaluation metrics may not capture all aspects of agent tool-use effectiveness. To reduce this threat, we report complementary metrics that evaluate both tool invocation correctness and final task success rate. Moreover, identifying information fields and constructing documentation variants involve manual inspection and correction, which may introduce subjective bias. To mitigate this threat, multiple authors independently conducted the analysis and resolved disagreements through discussion, achieving a Cohen's kappa coefficient of 0.862, with strong inter-rater agreement.

Finally, the inherent randomness of LLMs and the instability of agent execution pose additional threats to validity. To mitigate these threats, we keep the prompts and execution environments unchanged within each comparison, use the official replication artifacts of all baselines, and conduct three independent optimization runs, each followed by five evaluation runs. Thus, each evaluated setting is assessed over $3 \times 5 = 15$ runs. Across our evaluation, we apply the Mann-Whitney U test [31], and the improvements of DocsChisel remain statistically significant after Holm correction [16] ($p_{adj} < 0.05$).

VI. Related Work

A. LLM Agent Ecosystems

Recent advances in LLM agents and agent ecosystems have substantially improved the capability of LLMs to interact with external tools [3, 7, 15, 17, 25, 52]. Specifically, ReAct [56] introduces the reasoning-and-acting paradigm for iterative tool invocation, while Toolformer [44] demonstrates that LLMs can learn tool-use behaviors through self-supervised training. AutoGen [51] provides a unified framework for multi-agent orchestration and workflow automation. Subsequent works [27, 38, 40] further improve large-scale tool-use capabilities of LLM agents. Representative agent systems such as Claude Code [2], OpenAI Codex [36], and SWE-agent [54] have demonstrated the growing applicability of tool-use LLM agents in software engineering and other complex domains [28]. To evaluate the reliability and execution capability of tool-use LLM agents, recent studies [8, 21, 22, 23, 45, 48, 50, 53, 55, 57] propose diverse benchmarks that include tools with documentation.

Although these studies significantly advance tool-use capabilities of LLM agents, they mainly focus on tool construction, retrieval, invocation, and evaluation strategies, while largely treating tool documentation as fixed input for tool grounding and execution. In contrast, our work systematically investigates the structure and effectiveness of tool documentation itself, and studies how adaptive tool documentation optimization affects downstream task success rate across different agent settings.

B. Tool Documentation Engineering

Software engineering research has long recognized the importance of API usability, interface specifications, and documentation quality in supporting correct software usage [5, 6, 30, 32, 39, 49]. These works focus on API documentation generation [33], API documentation optimization [46] and documentation smell detection [24] to improve documentation readability, completeness, and consistency for human developers. With the rapid development of LLM agents, tool documentation has also attracted research attention. Patel et al. [9] automatically generate executable tool-use demonstrations to reduce the manual effort of writing tool documentation, while EasyTool [58] compresses existing tool documentation into concise instructions using predefined templates to reduce context overhead. A few works [10, 26, 42] further leverage LLMs and feedback from tool execution to iteratively refine the tool documentation for better tool invocation correctness.

These studies improve tool documentation quality and the tool-use capabilities of LLM agents to some extent. However, they mainly optimize tool documentation from an isolated, tool-level perspective, focusing on rewriting, correction, compression, or refinement within existing information fields. In contrast, DocsChisel does not treat documentation of each tool as an independent optimization target. Instead, it considers the interactions among tools within the same task domain and transfers optimization experience derived from domain-level and agent-aware execution failures to improve individual tool documentation by adding, removing, or refining information fields. In this way, DocsChisel aims not only to enhance the invocation correctness of each tool, but also to improve the overall task success of different LLM agents.

VII. Conclusion

We conduct an empirical study of tool documentation for LLM agents, revealing the varying effectiveness of different information fields on task success rate across task domains, LLM backbones, and agent paradigms. Our findings demonstrate that fixed tool documentation cannot consistently support different agent settings. Thus, we propose DocsChisel, an adaptive tool documentation optimization framework that automatically refines tool documentation based on the execution traces of LLM agents. Large-scale experiments have been conducted to demonstrate the effectiveness and efficiency of DocsChisel.

References

[1] Anonymous. (2026) Docschisel. Accessed on May 11, 2026. [Online]. Available: https://anonymous.4open.science/r/F42D-DocsChisel

[2] Anthropic. (2025) Claude code. Accessed on May 13, 2026. [Online]. Available: https://www.anthropic.com/claude-code

[3] ——. (2025) Writing effective tools for agents — with agents. Accessed on May 13, 2026. [Online]. Available: https://www.anthropic.com/engineering/writing-tools-for-agents

[4] ByteDance. (2024) Coze. Accessed on May 13, 2026. [Online]. Available: https://www.coze.com

[5] C. R. B. de Souza and D. L. M. Bentolila, "Automatic evaluation of api usability using complexity metrics and visualizations," in *Proceedings of the 31st International Conference on Software Engineering*, 2009, pp. 299–302.

[6] U. Dekel and J. D. Herbsleb, "Improving api documentation usability with knowledge pushing," in *Proceedings of the IEEE 31st International Conference on Software Engineering*, 2009, pp. 320–330.

[7] Dify.AI. (2024) Dify. Accessed on May 13, 2026. [Online]. Available: https://dify.ai

[8] Y. Du, F. Wei, and H. Zhang, "Anytool: Self-reflective, hierarchical agents for large-scale api calls," in *Proceedings of the 41st International Conference on Machine Learning*, 2024, pp. 11 812–11 829.

[9] W. Fang, Y. Zhang, K. Qian, J. Glass, and Y. Zhu, "Play2prompt: Zero-shot tool instruction optimization for llm agents via tool play," in *Proceedings of the Findings of the Association for Computational Linguistics*, 2025, pp. 26 274–26 290.

[10] S. Ghoshal, A. Mittal, J. Singh, M. Ballesteros, W. Sun, F. Tu, S. Singh, Y. Benajiba, F. Shah, S. Bharadwaj *et al.*, "Jtpro: A joint tool-prompt reflective optimization framework for language agents," *arXiv preprint arXiv:2604.19821*, 2026.

[11] T. Glm, A. Zeng, B. Xu, B. Wang, C. Zhang, D. Yin, D. Zhang, D. Rojas, G. Feng, H. Zhao *et al.*, "Chatglm: A family of large language models from glm-130b to glm-4 all tools," *arXiv preprint arXiv:2406.12793*, 2024.

[12] Google. (2023) Google gemini. Accessed on May 13, 2026. [Online]. Available: https://gemini.google.com/

[13] S. Gravitas. (2023) Autogpt: Build, deploy, and run ai agents. Accessed on May 13, 2026. [Online]. Available: https://github.com/significant-gravitas/autogpt

[14] R. Guo, K. Dong, X. Gao, and K. Das, "Learning to rewrite tool descriptions for reliable llm-agent tool use," *arXiv preprint arXiv:2602.20426*, 2026.

[15] J. He, C. Treude, and D. Lo, "Llm-based multi-agent systems for software engineering: Literature review, vision, and the road ahead," *ACM Transactions on Software Engineering and Methodology*, vol. 34, no. 5, pp. 1–30, 2025.

[16] S. Holm, "A simple sequentially rejective multiple test procedure," *Scandinavian Journal of Statistics*, vol. 6, no. 2, pp. 65–70, 1979.

[17] S. Hong, M. Zhuge, J. Chen, X. Zheng, Y. Cheng, J. Wang, C. Zhang, S. Yau, Z. Lin, L. Zhou *et al.*, "Metagpt: Meta programming for a multi-agent collaborative framework," in *Proceedings of the International Conference on Learning Representations*, vol. 2024, 2024, pp. 23 247–23 275.

[18] X. Hou, Y. Zhao, Y. Liu, Z. Yang, K. Wang, L. Li, X. Luo, D. Lo, J. Grundy, and H. Wang, "Large language models for software engineering: A systematic literature review," *ACM Transactions on Software Engineering and Methodology*, vol. 33, no. 8, pp. 1–79, 2024.

[19] C.-Y. Hsieh, S.-A. Chen, C.-L. Li, Y. Fujii, A. Ratner, C.-Y. Lee, R. Krishna, and T. Pfister, "Tool documentation enables zero-shot tool-usage with large language models," *arXiv preprint arXiv:2308.00675*, 2023.

[20] H. Huang, O. Zheng, D. Wang, J. Yin, Z. Wang, S. Ding, H. Yin, C. Xu, R. Yang, Q. Zheng *et al.*, "Chatgpt for shaping the future of dentistry: the potential of multi-modal large language model," *International Journal of Oral Science*, vol. 15, no. 1, p. 29, 2023.

[21] K.-H. Huang, A. Prabhakar, S. Dhawan, Y. Mao, H. Wang, S. Savarese, C. Xiong, P. Laban, and C.-S. Wu, "Crmarena: Understanding the capacity of llm agents to perform professional crm tasks in realistic environments," in *Proceedings of the Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies*, 2025, pp. 3830–3850.

[22] Y. Huang, J. Shi, Y. Li, C. Fan, S. Wu, Q. Zhang, Y. Liu, P. Zhou, Y. Wan, N. Gong *et al.*, "Metatool benchmark for large language models: Deciding whether to use tools and which to use," in *Proceedings of the International Conference on Learning Representations*, 2024, pp. 42 978–43 007.

[23] C. E. Jimenez, J. Yang, A. Wettig, S. Yao, K. Pei, O. Press, and K. Narasimhan, "Swe-bench: Can language models resolve real-world github issues?" in *Proceedings of the International Conference on Learning Representations*, vol. 2024, 2024, pp. 54 107–54 157.

[24] J. Y. Khan, M. Tawkat Islam Khondaker, G. Uddin, and A. Iqbal, "Automatic detection of five api documentation smells: Practitioners' perspectives," in *Proceedings of the IEEE International Conference on Software Analysis, Evolution and Reengineering*, 2021, pp. 318–329.

[25] LangChain. (2024) Langchain: Build agents faster, your way. Accessed on May 13, 2026. [Online]. Available: https://langchain.com

[26] H. Li, S. You, F. Di Palo, Y. Qian, and A. Jain, "Verification-guided context optimization for tool calling via hierarchical llms-as-editors," *arXiv preprint arXiv:2512.13860*, 2025.

[27] M. Li, Y. Zhao, B. Yu, F. Song, H. Li, H. Yu, Z. Li, F. Huang, and Y. Li, "Api-bank: A comprehensive benchmark for tool-augmented llms," in *Proceedings of the Conference on Empirical Methods in Natural Language Processing*, 2023, pp. 3102–3116.

[28] F. Lin, D. J. Kim, and T.-H. Chen, "Soen-101: Code generation by emulating software process models using large language model agents," in *2025 IEEE/ACM 47th International Conference on Software Engineering (ICSE)*, 2025, pp. 1527–1539.

[29] M. M. Liu, D. Garcia, F. Parllaku, V. Upadhyay, S. F. A. Shah, and D. Roth, "Toolscope: Enhancing llm agent tool use through tool merging and context-aware filtering," *arXiv preprint arXiv:2510.20036*, 2025.

[30] W. Maalej and M. P. Robillard, "Patterns of knowledge in api reference documentation," *IEEE Transactions on software Engineering*, vol. 39, no. 9, pp. 1264–1282, 2013.

[31] P. E. McKnight and J. Najab, "Mann-whitney u test," *The Corsini encyclopedia of psychology*, pp. 1–1, 2010.

[32] N. Nahar, S. Zhou, G. Lewis, and C. Kästner, "Collaboration challenges in building ml-enabled systems: Communication, documentation, engineering, and process," in *Proceedings of the 44th international conference on software engineering*, 2022, pp. 413–425.

[33] K. Nybom, A. Ashraf, and I. Porres, "A systematic mapping study on api documentation generation approaches," in *Proceedings of the 44th Euromicro Conference on Software Engineering and Advanced Applications*, 2018, pp. 462–469.

[34] OpenAI. (2023) Introducing chatgpt. Accessed on May 13, 2026. [Online]. Available: https://openai.com/blog/chatgpt

[35] OpenAI. (2024) Gpt-4o. Accessed on May 13, 2026. [Online]. Available: https://openai.com/index/hello-gpt-4o/

[36] ——. (2025) Introducing codex. Accessed on May 13, 2026. [Online]. Available: https://openai.com/index/introducing-codex/

[37] OpenAPIHub. (2024) Openapihub. Accessed on May 13, 2026. [Online]. Available: https://www.openapihub.com

[38] S. G. Patil, T. Zhang, X. Wang, and J. E. Gonzalez, "Gorilla: Large language model connected with massive apis," *Advances in Neural Information Processing Systems*, vol. 37, pp. 126 544–126 565, 2024.

[39] M. Piccioni, C. A. Furia, and B. Meyer, "An empirical study of api usability," in *Proceedings of the ACM / IEEE International Symposium on Empirical Software Engineering and Measurement*, 2013, pp. 5–14.

[40] Y. Qin, S. Liang, Y. Ye, K. Zhu, L. Yan, Y. Lu, Y. Lin, X. Cong, X. Tang, B. Qian *et al.*, "Toolllm: Facilitating large language models to master 16000+ real-world apis," in *Proceedings of the International Conference on Learning Representations*, vol. 2024, 2024, pp. 9695–9717.

[41] C. Qu, S. Dai, X. Wei, H. Cai, S. Wang, D. Yin, J. Xu, and J.-R. Wen, "Towards completeness-oriented tool retrieval for large language models," in *Proceedings of the 33rd ACM International Conference on Information and Knowledge Management*, 2024, pp. 1930–1940.

[42] ——, "From exploration to mastery: Enabling llms to master tools via self-driven interactions," in *Proceedings of the International Conference on Learning Representations*, 2025, pp. 55 675–55 697.

[43] RapidAPI. (2024) Rapidapi. Accessed on May 13, 2026. [Online]. Available: https://rapidapi.com

[44] T. Schick, J. Dwivedi-Yu, R. Dessì, R. Raileanu, M. Lomeli, E. Hambro, L. Zettlemoyer, N. Cancedda, and T. Scialom, "Toolformer: Language models can teach themselves to use tools," *Advances in Neural Information Processing Systems*, vol. 36, pp. 68 539–68 551, 2023.

[45] H. Shen, Y. Li, D. Meng, D. Cai, S. Qi, L. Zhang, M. Xu, and Y. Ma, "Shortcutsbench: A large-scale real-world benchmark for api-based agents," in *Proceedings of the International Conference on Learning Representations*, 2025, pp. 77 772–77 797.

[46] L. Shi, H. Zhong, T. Xie, and M. Li, "An empirical study on evolution of api documentation," in *Proceedings of the International Conference on Fundamental Approaches To Software Engineering*, 2011, pp. 416–431.
[47] Z. Shi, S. Gao, L. Yan, Y. Feng, X. Chen, Z. Chen, D. Yin, S. Verberne, and Z. Ren, "Tool learning in the wild: Empowering language models as automatic tool agents," in *Proceedings of the ACM on Web Conference*, 2025, pp. 2222–2237.
[48] O. Styles, S. Miller, P. Cerda-Mardini, T. Guha, V. Sanchez, and B. Vidgen, "Workbench: a benchmark dataset for agents in a realistic workplace setting," *arXiv preprint arXiv:2405.00823*, 2024.
[49] S. Subramanian, L. Inozemtseva, and R. Holmes, "Live api documentation," in *Proceedings of the 36th international conference on software engineering*, 2014, pp. 643–652.
[50] Q. Tang, Z. Deng, H. Lin, X. Han, Q. Liang, B. Cao, and L. Sun, "Toolalpaca: Generalized tool learning for language models with 3000 simulated cases," *ArXiv*, vol. abs/2306.05301, 2023.
[51] Q. Wu, G. Bansal, J. Zhang, Y. Wu, B. Li, E. Zhu, L. Jiang, X. Zhang, S. Zhang, J. Liu *et al.*, "Autogen: Enabling next-gen llm applications via multi-agent conversations," in *Proceedings of the 1st Conference on Language Modeling*, 2024.
[52] C. S. Xia, Y. Deng, S. Dunn, and L. Zhang, "Demystifying llm-based software engineering agents," *Proceedings of the ACM on Software Engineering*, vol. 2, no. FSE, pp. 801–824, 2025.
[53] Q. Xu, F. Hong, B. Li, C. Hu, Z. Chen, and J. Zhang, "On the tool manipulation capability of open-source large language models," *ArXiv*, vol. abs/2305.16504, 2023.
[54] J. Yang, C. E. Jimenez, A. Wettig, K. Lieret, S. Yao, K. Narasimhan, and O. Press, "Swe-agent: Agent-computer interfaces enable automated software engineering," *Advances in Neural Information Processing Systems*, vol. 37, pp. 50 528–50 652, 2024.
[55] S. Yao, N. Shinn, P. Razavi, and K. Narasimhan, "τ-bench: A benchmark for tool-agent-user interaction in real-world domains," *arXiv preprint arXiv:2406.12045*, 2024.
[56] S. Yao, J. Zhao, D. Yu, N. Du, I. Shafran, K. Narasimhan, and Y. Cao, "React: Synergizing reasoning and acting in language models," *arXiv preprint arXiv:2210.03629*, 2022.
[57] P. Yu, W. Liu, Y. Yang, J. Li, Z. Zhang, X. Feng, and F. Zhang, "Benchmarking llm tool-use in the wild," *arXiv preprint arXiv:2604.06185*, 2026.
[58] S. Yuan, K. Song, J. Chen, X. Tan, Y. Shen, K. Ren, D. Li, and D. Yang, "Easytool: Enhancing llm-based agents with concise tool instruction," in *Proceedings of the 2025 Conference of the Nations of the Americas Chapter of the Association for Computational Linguistics: Human Language Technologies (Volume 1: Long Papers)*, 2025, pp. 951–972.
[59] A. Zeng, X. Lv, Z. Hou, Z. Du, Q. Zheng, B. Chen, D. Yin, C. Ge, C. Huang, C. Xie *et al.*, "Glm-5: from vibe coding to agentic engineering," *arXiv preprint arXiv:2602.15763*, 2026.
[60] L. Zhuang, Q. Zhang, H. Zhou, Y. Zhang, and X. Huang, "Losemb: Logic-guided semantic bridging for inductive tool retrieval," in *Proceedings of the ACM Web Conference*, 2026, pp. 3835–3846.